zkr-staking Migration Contracts

Zircuit Labs

HALBORN

zkr-staking Migration Contracts - Zircuit Labs

Prepared by: HALBORN

Last Updated 08/07/2024

Date of Engagement by: July 31st, 2024 - August 2nd, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	0	0	0	0	7

TABLE OF CONTENTS

1. Introduction

`Zircuit Labs` engaged `Halborn` to conduct a security assessment of their `zkr-staking` project **beginning on July 31st and ending on August 2nd**. The security assessment was scoped to the smart contracts provided in the GitHub repository. Commit hash and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 3 days for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified seven informational security findings that were acknowledged by the `Zircuit Labs team`.

It is important to note that most state-changing functions of the contracts in scope are protected, so only authorized addresses can call them. Special mention to the `withdrawUnclaimedFunds()` of the `ZrcDistributor` contract, which would allow the default admin to withdraw all tokens from it. The `Zircuit Labs team` indicated that they will assign the default admin role to a multi-sig with operational best practices to ensure the role is not compromised. Aside from this compromise, there is a low probability for abuse of the `withdrawUnclaimedFunds()` function since the `Zircuit Labs team` itself will fund the contract with the ZRC tokens. Furthermore, there will be multiple instances of the `ZRCDistributor` contract deployed with only one active instance at a time. This limits the damage of a compromised key to just the ZRC tokens in that one instance of `ZRCDistributor`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (`MythX`)
- Static Analysis of security for scoped contract, and imported functions (`slither`).
- Testnet deployment (`Foundry`).

Out-Of-Scope

- External libraries and financial-related attacks.
- New features/implementations after/with the **remediation commit IDs**. The review of the new cancellation mechanism and the deployment scripts was later conducted, uncovering two informational risk findings.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: zkr-staking

(b) Assessed Commit ID: 3630169

(c) Items in scope:

- contracts/BatchMigrator.sol
- contracts/Migrator.sol
- contracts/ZrcDistributor.sol

Out-of-Scope: Third party dependencies.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
USE OF UNSAFE ERC20 OPERATIONS	INFORMATIONAL	ACKNOWLEDGED
UNPROTECTED PRIVILEGED ROLES	INFORMATIONAL	ACKNOWLEDGED
UNUSED FUNCTION ARGUMENTS	INFORMATIONAL	ACKNOWLEDGED
CONSIDER USING NAMED MAPPINGS	INFORMATIONAL	ACKNOWLEDGED
LACK OF EVENT EMISSION	INFORMATIONAL	ACKNOWLEDGED
GLOBAL STATE VARIABLES COULD BE IMMUTABLE	INFORMATIONAL	ACKNOWLEDGED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
USE OF UNLICENSED SMART CONTRACTS	INFORMATIONAL	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 USE OF UNSAFE ERC20 OPERATIONS

// INFORMATIONAL

Description

During the security assessment, it was found that the `migrate()` function of the contract `Migrator` is using the unsafe ERC20 function `transferFrom()` instead of its safer wrapper version from OpenZeppelin's `SafeERC20`: `safeTransferFrom()`.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:P/S:U (1.3)

Recommendation

It is recommended to use OpenZeppelin's `SafeERC20`'s `safeTransferFrom()` function instead of `transferFrom()`.

Remediation Plan

ACKNOWLEDGED: The **Zircuit team** acknowledged this finding by indicating "Since all tokens being transferred are known entities and do not use unconventional transfer operations, we can safely avoid using `safeTransfer()` to reduce gas costs".

7.2 UNPROTECTED PRIVILEGED ROLES

// INFORMATIONAL

Description

It was observed that the `ZrcDistributor` contract inherited from the well-known `AccessControl` library from OpenZeppelin. The implementation is making use of the default `DEFAULT_ADMIN_ROLE` privilege which can be easily renounced leaving the privileged functions unusable. The contract is also not implementing a two-step mechanism for transferring the `DEFAULT_ADMIN_ROLE` privilege.

Proof of Concept

The following PoC can be executed to show how the two privileged roles can be renounced in one call:

```
it("Should renounce role", async function () {
    const { zrcDistributor, multisig, claimer } = await loadFixture(zrcDistributorFixture)

    expect(await zrcDistributor.hasRole(await zrcDistributor.DEFAULT_ADMIN_ROLE(), multisig)).to
    await zrcDistributor.connect(multisig).renounceRole(await zrcDistributor.DEFAULT_ADMIN_ROLE(
    expect(await zrcDistributor.hasRole(await zrcDistributor.DEFAULT_ADMIN_ROLE(), multisig)).to

    expect(await zrcDistributor.hasRole(await zrcDistributor.CLAIMER_ROLE(), claimer)).to.be.tru
    await zrcDistributor.connect(claimer).renounceRole(await zrcDistributor.CLAIMER_ROLE(), clai
    expect(await zrcDistributor.hasRole(await zrcDistributor.CLAIMER_ROLE(), claimer)).to.be.fal
});
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:P/S:U (1.3)

Recommendation

As recommended by OpenZeppelin, consider using `AccessControlDefaultAdminRules` instead for an extra layer of security. This contract implements the following risk mitigations on top of `AccessControl`:

- Only one account holds the `DEFAULT_ADMIN_ROLE` since deployment until it's potentially renounced.
- Enforces a 2-step process to transfer the `DEFAULT_ADMIN_ROLE` to another account.
- Enforces a configurable delay between the two steps, with the ability to cancel before the transfer is accepted.
- The delay can be changed by scheduling, see `changeDefaultAdminDelay`.
- It is not possible to use another role to manage the `DEFAULT_ADMIN_ROLE`.

> ⚠ WARNING
>
> The `DEFAULT_ADMIN_ROLE` is also its own admin: it has permission to grant and revoke this role. Extra precautions should be taken to secure accounts that have been granted it. We recommend using `AccessControlDefaultAdminRules` to enforce additional security measures for this role.

Remediation Plan

ACKNOWLEDGED: The **Zircuit team** acknowledged this finding by indicating "The lifetime of ZRCDistributor will be short as it will only be used for a subset of ZRC token claims within a certain time period. New ZRCDistributor contracts will be deployed as needed for additional token claims. Thus, it will be unnecessary to transfer ownership of the contract. There is also no need to renounce the admin role as it will be useless once the ZRC in the contract has been depleted, and so the renouncing of the role will never happen".

7.3 UNUSED FUNCTION ARGUMENTS

// INFORMATIONAL

Description

The `migrate()` function of the `Migrator` contract contains two arguments (`_user` and `_destination`) that are never used. This makes the functions `batchMigrate()` and `batchMigrateWithDestinationEqualToUser()` of the `BatchMigrator` contract to always have the same outcome.

```
function migrate(
  address _user, // @audit unused argument
  address[] calldata _tokens,
  address _destination, // @audit unused argument
  uint256[] calldata _amounts
) external {
```

Notice that the affected `migrate()` function will always be called by the `ZtakingPool` contract (not in scope) which will validate the `_user` and `_destination` parameters.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:F/S:U (0.1)

Recommendation

Consider removing the unused argument names and documenting the reason for not using them.

Remediation Plan

ACKNOWLEDGED: The **Zircuit team** acknowledged this finding by indicating "The signature of the `Migrator.migrate()` function was determined before the code of the contract was written. This signature was fixed in the ZtakingPool contract, which had been previously deployed. At the time, it was anticipated that these function parameters were needed. However, due to design changes, they were no longer needed in the final implementation of the Migrator contract".

7.4 CONSIDER USING NAMED MAPPINGS

// INFORMATIONAL

Description

The project is using a Solidity version greater than `0.8.18`, which supports named mappings. Using named mappings can improve the readability and maintainability of the code by making the purpose of each mapping clearer. This practice helps developers and auditors understand the mappings' intent more easily.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Consider refactoring the mappings to use named arguments, which will enhance code readability and make the purpose of each mapping more explicit.
For example, on `ZrcDistributor.sol`, instead of declaring:

```
mapping(address => bool) public claimed;
```

It could be declared as:

```
mapping(address account => bool claimed) public claimed;
```

Remediation Plan

ACKNOWLEDGED: The **Zircuit team** acknowledged this finding by indicating "We agree that named mappings increase readability. Since there is no security impact, we will not make this change. However, we will keep this in mind for any future code written".

7.5 LACK OF EVENT EMISSION

// INFORMATIONAL

Description

It has been observed that some functionalities are missing emitting events.
Events are a method of informing the transaction initiator about the actions taken by the called function. It logs its emitted parameters in a specific log history, which can be accessed outside of the contract using some filter parameters. Events help non-contract tools to track changes, and events prevent users from being surprised by changes.
Example:

- In `BatchMigrator.sol`, the `setSignatureExpiry()` does not emit any event.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

All functions updating important parameters should emit events.

Remediation Plan

ACKNOWLEDGED: The **Zircuit team** acknowledged this finding by indicating "Since setting the signature expiry will most likely be a one-time operation, monitoring a corresponding event will not be useful".

7.6 GLOBAL STATE VARIABLES COULD BE IMMUTABLE

// INFORMATIONAL

Description

Global state variables that are set in the constructor and never updated again should be declared as `immutable` to save gas.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to add the `immutable` attribute to the `token` and `merkleRoot` state variables in the `ZrcDistributor` contract.

Remediation Plan

ACKNOWLEDGED: The **Zircuit team** acknowledged this finding by indicating "We acknowledge that the suggestion would lead to improved gas efficiency. However, since this contract will be deployed on Zircuit, the additional gas cost will be minimal".

7.7 USE OF UNLICENSED SMART CONTRACTS

// INFORMATIONAL

Description

All the zkr-staking smart contracts are marked as unlicensed, as indicated by the SPDX license identifier at the top of the files:

```
// SPDX-License-Identifier: UNLICENSED
```

Using unlicensed contract can lead to legal uncertainties and potential conflicts regarding the usage, modification and distribution rights of the code. This may deter other developers from using or contributing to the project and could potentially lead to legal issues in the future.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is strongly recommended to choose and apply an appropriate open-source license to the smart contract. Some popular options for blockchain and smart contract projects include:
1. MIT License: A permissive license that allows for reuse with minimal restrictions.
2. GNU General Public License (GPL): A copyleft license that ensures derivative works are also open-source.
3. Apache License 2.0: A permissive license that provides an express grant of patent rights from contributors to users.

Remediation Plan

ACKNOWLEDGED: The **Zircuit team** acknowledged this finding by indicating "We do not wish to add a license at this time and are not concerned with the legal impact of reusing this code".

STATIC ANALYSIS REPORT

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Output

```
INFO:Detectors:
Migrator.migrate(address,address[],address,uint256[]) (contracts/Migrator.sol#25-42) uses arbitrary from in transferFrom: IERC20(_tokens[i]).transferFrom(
ztakingPool,zircuitMultisig,_amounts[i]) (contracts/Migrator.sol#36-40)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
Migrator.migrate(address,address[],address,uint256[]) (contracts/Migrator.sol#25-42) ignores return value by IERC20(_tokens[i]).transferFrom(ztakingPool,z
ircuitMultisig,_amounts[i]) (contracts/Migrator.sol#36-40)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.